# RESOURCE MANAGEMENT IN A MULTI-PROCESSOR SYSTEM

The present invention relates to a resource management method and apparatus that is particularly, but not exclusively, suited to resource management of multi-processor real-time systems.

For systems-on-silicon (SoS) or systems-on-chip (SoC), memory is becoming a dominant limiting factor, since, from the point of view of the amount of silicon needed, adding another processing core is no longer a problem. As a consequence, a SoS or SoC may contain multiple processor cores.

The management of memory is a crucial aspect of resource management for a multiprocessor system. Various methods have been developed to optimize memory use for single processor systems, including generalizing the use of preemption points to the management of main memory, especially in real-time systems. In these approaches, rather than preempting tasks at arbitrary moments during their execution, those tasks are preferably only preempted at dedicated preemption points based on their memory usage.

In a typical prior art approach, suspension of a task is referred to as task preemption, or preemption of a task, and the term "task" is used to denote a unit of execution that can compete on its own for system resources such as memory, CPU, I/O devices, etc. For purposes of discussion, a task is assumed to be a succession of continually executing jobs, each of which comprises one or more sub-jobs. For example, a task can comprise "demultiplexing a video stream", and involve reading-in incoming streams, processing the streams and outputting corresponding data. These steps are carried out with respect to each incoming data stream, so that reading, processing and outputting with respect to a single stream corresponds to performing one job, each having three sub-jobs. Thus, when there is a plurality of packets of data to be read-in and processed, the job would be performed a corresponding plurality of times. A sub-job can be considered to relate to a functional component of the job and in a multiprocessor system each stream would be assigned to a different processor or subset of processors.

A known method of scheduling a plurality of tasks in a data processing system requires that each sub-job of a task have a set of suspension criteria, called suspension data, that specifies the processing preemption points and corresponding conditions for suspension of a sub-job based on its memory usage [4] [5]. The amount of memory that is used by the data processing system is thus indirectly controlled by this suspension data, via these preemption points, which specify the amounts of memory required at these preemption points in a job's execution.

Thus, these preemption points can be utilized to avoid data processing system crashes due to a lack of memory. When a real-time task is characterized as comprising a plurality of sub-jobs, its preemption points preferably coincide with the sub-job boundaries of the task.

Data indicative of memory usage of a task conforming to the suspension data associated with each sub-job of a task can, for example, be embedded into a task via a line of code that requests a descheduling event, specifying that a preemption point has been reached in the processing of the task, i.e., a sub-job boundary has been reached. That is, the set of start points of the sub-jobs of a task constitute a set of preemption points of that task. The $j^{th}$ preemption point $P_{i,j}$ of a task $\tau_i$ is characterized by information related to the preemption point itself and information related to the succeeding non-preemptible sub-job interval $I_{i,j}$ between the $j^{th}$ preemption point and the next preemption point, i.e., the $(j+1)^{th}$ preemption point.

At run time, a task informs the controlling operating system when it arrives at preemption points, e.g. when it starts a sub-job, switches between sub-jobs, and completes a sub-job, and the operating system decides when and where execution of a task is preempted. Ideally, preemption may occur at a preemption point or at any other point during the execution of a task. However, such flexibility of choice of preemption comes at the cost of consistency so that preemption is limited to preemption points to maintain consistency.

A prior art preemption point approach based on main memory requirements that does not jeopardize consistency of the system, necessarily limits the preemption of all tasks to their preemption points and matching synchronization primitives for controlling exclusive use of a resource to both be within a sub-job boundary. As is known in the art, a component (e.g. a software component, which can comprise one or more tasks) can have a programmable interface that comprises the properties, functions or methods and events that the component defines [6]. For purposes of discussion, a task is assumed to be accompanied by an interface

100 that includes, at a minimum, main memory data required by the task, $MP_{i,J}$ 101b, as illustrated in FIG. 1.

For the purposes of discussion, a task is assumed to be periodic and real-time, and characterized by a period T and a phasing F, where 0 <= F < T, which means that a task comprises a sequence of sub-jobs, each of which is released at time F+nT, where n = 0 ...N. As an example only and as illustrated in FIG. 2, set-top box 200 is assumed to execute three tasks – (1) display menu on the User Interface 205, (2) retrieve text information from a content provider 203, and (3) process some video signals – and each these 3 tasks is assumed to comprise a plurality of sub-jobs. For ease of presentation, it is assumed that the sub-jobs are executed sequentially.

At least some of these sub-jobs can be preempted and the boundaries between these sub-jobs that can be preempted provide preemption points and are summarized in Table 1:

| Task $\tau_i$ | Task description | Number of preempt-able sub-jobs of task $\tau_i$ m(i) |
|---|---|---|
| $\tau_1$ | display menu on the GUI | 3 |
| $\tau_2$ | retrieve text information from content provider | 2 |
| $\tau_3$ | process video signals | 2 |

TABLE 1

Referring also to FIG.3, for each task the suspension data 101 comprises: information relating to a preemption-point $P_{i,j}$ 301, such as the maximum amount of memory $MP_{i,j}$ 302 required at the preemption point, and information relating to the interval $I_{i,j}$ 303 between successive preemption-points, such as the worst-case amount of memory $MI_{i,j}$ 304 required in an intra-preemption point interval (i represents task $\tau_i$ and j represents a preemption point).

More specifically, suspension data 101 comprises data specifying

1. preemption point j of the task $\tau_i$ ($P_{i,j}$) 101a;

2. maximum memory requirements of task $\tau_i$, $MP_{i,j}$, at preemption point j of that task, where $1 \leq j \leq m(i)$ 101b;
3. interval, $I_{i,j}$, between successive preemption points j and (j+1) corresponding to sub-job j of task $\tau_i$, where $1 \leq j \leq m(i)$ 101c; and
4. maximum (i.e. worst-case) memory requirements of task $\tau_i$, $MI_{i,j}$, i n the interval j of that task, where $1 \leq j \leq m(i)$ 101d.

Table 2 illustrates the suspension data 101 for the current example (each task has its own interface, so that in the current example, the suspension data 101 corresponding to the first task $\tau_1$ compri ses the data in the first row of Table 2, the suspension data 101 corresponding to the second task $\tau_2$ comprises the second row of Table 2, etc.):

| Task $\tau_i$ | $MP_{i,1}$ | $MI_{i,1}$ | $MP_{i,2}$ | $MI_{i,2}$ | $MP_{i,3}$ | $MI_{i,3}$ |
|---|---|---|---|---|---|---|
| $\tau_1$ | 0.2 | 0.6 | 0.2 | 0.4 | 0.1 | 0.6 |
| $\tau_2$ | 0.1 | 0.5 | 0.2 | 0.8 | - | - |
| $\tau_3$ | 0.1 | 0.2 | 0.1 | 0.3 | - | - |

TABLE 2

Suppose a set-top box 200 is equipped with 1.5 Mbytes of memory. Under normal, or non-memory based preemption conditions, this set-top box 200 behaves as follows.

Referring now to FIG. 4A, a processor 401 may be expected to schedule tasks according to some sort of time slicing or priority based preemption, meaning that all 3 tasks run concurrently, i.e. effectively at the same time. It is therefore possible that each task can be scheduled to run its most memory intensive sub-job at the same time. The worst-case memory requirement of these three tasks, $M^P$, is given by:

$$M^P = \sum_{i=1}^{3} \max_{j=1}^{m(i)} MI_{i,j} \qquad \text{(Equation 1)}$$

For tasks $\tau_1$, $\tau_2$ and $\tau_3$ $M^P$ is thus the maximum memory requirements of $\tau_1$ (being $MI_{1,1}$) plus the maximal memory requirements of task $\tau_2$ (being $MI_{2,2}$) plus the maximal memory

requirements of task $\tau_3$ (being $MI_{3,2}$). These maximum requirements are indicated by the Table 2 entries in bold:

$$M^P = 0.6 + 0.8 + 0.3 = 1.7 \text{ Mbytes.}$$

This exceeds the memory available to the set-top box 200 by 0.2 Mbytes, so that, in the absence of any precautionary measures and if these sub-jobs are to be processed at the same time, the set-top box 200 crashes.

Referring now to FIG. 5, suppose tasks are scheduled in accordance with a scheduling algorithm and a data structure is maintained for each task $\tau_i$ after it has been created and that matching pairs of primitives for exclusive use of a resource do not span a sub-job boundary. Suppose further that a scheduler 501 employs a conventional priority-based, preemptive scheduling algorithm, which essentially ensures that, at any point in time, the currently running task is the one with the highest priority among all ready-to-run tasks in the system. As is known in the art, the scheduling behavior can be modified by selectively enabling and disabling preemption for the running, or ready-to-run, tasks.

A task manager 503 receives the suspension data 101 corresponding to a newly received task and evaluates whether preemption is required or not and if it is required, passes this newly received information to the scheduler 501, requesting preemption. Suppose details of the tasks are as defined in Table 2, and assume that task $\tau_1$ (and only $\tau_1$) is currently being processed and that the scheduler is initially operating in a mode in which there are no memory-based constraints.

Suppose now that task $\tau_2$ is received by the task manager 503, which reads the suspension data 101 from its interface $Int_2$ 100, and identifies whether or not the scheduler 501 is working in accordance with memory-based preemption. Since, in this example, it is not, the task manager 503 evaluates whether the scheduler 501 needs to change to memory-based preemption. This therefore involves the task manager 503 retrieving worst case suspension data corresponding to all currently executing tasks (in this example task $\tau_1$) from a suspension data store 505, evaluating Equation 1 and comparing the evaluated worst-case memory

requirements with the memory resources available. Continuing with the example introduced in Table 2, Equation 1, for $\tau_1$ and $\tau_2$, is:

$$M^P = \sum_{i=1}^{2} \max_{j=1}^{m(i)} MI_{i,j} = 0.6 + 0.8 = 1.4 \text{ Mbytes}$$

This is less than the available memory, so there is no need to change the mode of operation of the scheduler 501 to memory-based preemption (i.e. there is no need to constrain the scheduler based on memory usage). Thus, if the scheduler 501 were to switch between task $\tau_1$ and task $\tau_2$ – e.g. to satisfy execution time constraints of task $\tau_2$, meaning that both tasks effectively reside in memory at the same time – the processor never accesses more memory than is available.

Next, and before tasks $\tau_1$ and $\tau_2$ have completed, another task $\tau_3$ is received. The task manager 503 reads the suspension data 101 from interface $Int_3$ associated with the task $\tau_3$, evaluating whether the scheduler 501 needs to change to memory-based preemption. Assuming that the scheduler 501 is multi-tasking tasks $\tau_1$ and $\tau_2$, the worst case memory requirements for all three tasks is now

$$M^P = \sum_{i=1}^{3} \max_{j=1}^{m(i)} MI_{i,j} = 0.6 + 0.8 + 0.3 = 1.7 \text{ Mbytes}$$

This exceeds the available memory, so the task manager 503 requests and retrieves memory usage data $MP_{i,j}$,$MI_{i,j}$ 101b, 101d for all three tasks from the suspension data store 505, and evaluates whether, based on this retrieved memory usage data, there are sufficient memory resources to execute all three tasks. This can be ascertained through evaluation of the following equation:

$$M^D = \sum_{i=1}^{3} \max_{j=1}^{m(i)} MP_{i,j} + \max_{i=1}^{3} (\max_{j=1}^{m(i)} MI_{i,j} - \max_{j=1}^{m(i)} MP_{i,j}) \quad \text{(Equation 2)}$$

$$= 0.2 + 0.2 + 0.1 + \max(0.6 - 0.2, 0.8 - 0.2, 0.3 - 0.1)$$

$$= 0.5 + 0.6 = 1.1 \text{ Mbytes.}$$

This memory requirement is lower than the available memory, meaning that, provided the tasks are preempted based on their memory usage, all three tasks can be executed concurrently.

Accordingly, the task manager 503 invokes "memory-based preemption mode" by instructing the tasks to transmit deschedule instructions to the scheduler 501 at their designated preemption points $MP_{i,j}$. In this mode, the scheduler 501 allows each task to run non-preemptively from one preemption point to the next, with the constraint that, at any point in time, at most one task at a time can be at a point other than one of its preemption points. Assuming that the newly arrived task starts at a preemption point, the scheduler 501 ensures that this condition holds for the currently running tasks, thereby constraining all but one task to be at a preemption point.

Thus, in one known memory-based preemption mode, the scheduler 501 is only allowed to preempt tasks at their memory preemption points (i.e. in response to a deschedule request from the task at their memory-based preemption points).

When one of the tasks has terminated, the terminating task informs the task manager 503 that it is terminating, causing the task manager 503 to evaluate Equation 1 and if the worst case memory usage (taking into account removal of this task) is lower than that available to the scheduler 501, the task manager 503 can cancel memory-based preemption, which has the benefit of enabling the system to react faster to external events (since the processor is no longer "blocked" for the duration of the sub-jobs). In general, termination of a task is typically caused by its environment, e.g. a switch of channel by the user or a change in the data stream of the encoding applied (requiring another kind of decoding), meaning that the task manager 503 and/or scheduler 501 should be aware of the termination of a task and probably even instruct the task to terminate.

It should be noted that, when invoked, memory-based preemption constraints are obligatory.

The tasks have been described as software tasks, but a task can also be implemented in hardware. Typically, a hardware device (behaving as a hardware task) is controlled by a software task, which allocates the (worst-case) memory required by the hardware device, and subsequently instructs the hardware task to run. When the hardware task completes, it informs

the software task, which subsequently de-allocates the memory. Hence, by having a controlling software task, hardware tasks can simply be dealt with as described above.

This approach applies to a single processor system and is not optimized for a multi-processor system and thus a multi-processor optimization is needed.

The present invention provides optimizations of the single processor approach to memory based resource management described above, for multi-processor systems. Consider a set $\Gamma$ of n tasks $\tau_i$ ($1 \leq i \leq n$) and a set P of p processors $\pi_k$ ($1 \leq k \leq p$), where n is typically much larger than p and assume a fixed-priority preemptive scheduling (FPPS) scheme for the non-constrained mode, and a fixed-priority scheduling scheme with deferred preemption (FPDP) scheme for the memory-based preemption mode.

There are three alternative preferred embodiments, depending on the allocation of tasks to processors:

1. Fixed Allocation: every task $\tau_i$ is allocated to a particular processor $\pi_k$, i.e. task $\tau_i$ will exclusively execute on processor $\pi_k$. This embodiment is preferred when processors are dedicated, i.e., where each processor differs essentially from every other processor.
2. Variable Allocation: every task $\tau_i$ may execute on every processor $\pi_k$. At run-time the scheduler determines which processor executes which task. A task may be preempted while running on one processor, and later continue on another. This embodiment is preferred when all the processors are identical.
3. Mixed Allocation: every task $\tau_i$ is allocated to a subset of processors. This is a natural approach when the set of processors can be divided into subsets in which the processors are identical.

The foregoing and other features and advantages of the invention will be apparent from the following, more detailed description of preferred embodiments as illustrated in the accompanying drawings in which reference characters refer to the same parts throughout the various views.

FIG. 1 illustrates a schematic diagram of components of a task interface according to an embodiment of the present invention;

FIG. 2 illustrates a schematic diagram of an example of a digital television system in which an embodiment of the present invention is operative;

FIG. 3 illustrates a schematic diagram of the relationships between components of the task interface illustrated in FIG. 1 for a single processor.

FIG. 4A illustrates components constituting the set-top; box of FIG. 2, for a single processor system.

FIG. 4B illustrates components constituting the set-top box of FIG. 2 for a multiprocessor system.

FIG. 5 illustrates components of the processor of the set-top box illustrated in FIG. 2 and FIG. 4A.

It is to be understood by persons of ordinary skill in the art that the following descriptions are provided for purposes of illustration and not for limitation. An artisan understands that there are many variations that lie within the spirit of the invention and the scope of the appended claims. Unnecessary detail of known functions and operations may be omitted from the current description so as not to obscure the present invention.

High volume electronic (HVE) consumer systems, such as digital TV sets, digitally improved analog TV sets and set-top boxes (STBs) must provide real-time services while remaining cost-effective and robust. Consumer products, by their nature, are heavily resource constrained. As a consequence, the available resources have to be used very efficiently, while preserving typical qualities of HVE consumer systems, such as robustness, and meeting stringent timing requirements. Concerning robustness, no one expects, for example, a TV set to fail with the message "please reboot the system".

Significant parts of the media processing in HVE consumer systems are implemented in on-board software that handles multiple concurrent streams of data, and in particular must very efficiently manage system resources, such as main memory, in a multi-tasking environment. Consider a set-top box as an example of an HVE consumer system requiring real-time resource management. Conventionally, as illustrated in FIG. 2, a set-top box 200 receives input for television 201 from a content provider 203 (a server or cable) and from a user interface 205. The user interface 205 comprises a remote control interface for receiving signals from a user-controlled remote device 202, e.g., a handheld infrared remote transmitter.

The set-top box 200 receives at least one data stream from at least one of an antenna and a cable television outlet, and performs at least one of processing the data stream or forwarding the data stream to television 201. A user views the at least one data stream displayed on television 201 and via user interface 205, makes selections based on what is being displayed. The set-top box 200 processes the user selection input and based on this input may transmit to the content provider 203 the user input, along with other information identifying the set-top 200 and its capabilities.

FIG. 4B illustrates a simplified block diagram of an exemplary system 450 of a typical set-top box 200 that may include a plurality of processors 460.1 - 460.3 managed by a control and allocation logic module which allocates tasks to processors 460.1 - 460.3 and controls the overall operation of set-top box 200. The control and allocation logic module 451 comprises a data receiver 452 for receiving put data from the network 407 and from storage 406, an evaluator 453 for evaluating memory usage, an allocator 454 for allocating tasks to processors, a selector 455 for selecting tasks to initiate and terminate execution thereof and a scheduler 501 for scheduling tasks for execution and descheduling executing tasks. The control and allocation logic module 451 is coupled to a television tuner 403, a memory 405, a long term storage device 406, a communication interface 407, and a remote interface 409. The television tuner 403 receives television signals over transmission line 411 and these signals may originate from at least one of an antenna (not shown) and a cable television outlet (not shown). The control & allocation logic module 451 manages the user interface 205, providing data, audio and video output to the television 201 via line 413. The remote interface 409 receives signals from the remote control via the wireless connection 415. The communication interface 407 interfaces between the set-top box 200 and at least one remote processing system, such a Web server, via data path 417. The communication interface 417 is at least one of a telephone modem, an Integrate Services Digital Network (ISDN) adapter, a Digital Subscriber Line (xDSL), a cable television modem, and any other suitable data communication device. The exemplary system 450 of FIG. 4B is for descriptive purposes only. Although the description may refer to terms commonly used in describing particular set-top boxes 200, the description and concepts equally apply to other control processors, including systems having architectures dissimilar to that shown in FIG. 4B.

The control and allocation logic module 451, in a preferred embodiment, is configured to allocate a plurality of real-time tasks relating to the control of the set-top box 200 to a plurality of multi-processors 460.1 - 460.3 that share a memory 405. These real-time tasks include changing channels, selection of a menu option displayed on the user interface 205, decoding incoming data streams, recording incoming data streams using the long term storage device 406 and replaying them, etc. The operation of the set-top box is determined by these real-time control tasks based on characteristics of the set-top box 100, incoming video signals via line 411, user inputs via user interface 205, and any other ancillary input.

In a preferred embodiment, multi-processors share memory 405. These multi-processors 460.1 - 460.3 may each be CPUs, or co-processors, or other processing devices. In a preferred embodiment, the same task is never executed simultaneously by two (or more) processors. In a preferred embodiment, the control and allocation logic module comprises a single task-manager for the entire set of processors 460.1 - 460.3. This is a centralized approach for discussion purposes only. The present invention is not constrained to centralized approaches, and decentralized versions may be easily conceived by individuals experienced in the art.

Whenever every task $\tau$ is allocated to a particular processor $\pi$, the set $\Gamma$ of tasks may be partitioned into p disjoint sets $\Gamma_1$ - $\Gamma_p$, where the subset $\Gamma_k$ is allocated to processor $\pi_k$. The maximum amount of memory required by the subset of tasks $\Gamma_k$ under FPPS as executed by processor $\pi_k$ is given by (Eq. 1s'), which is a variant of (Eq. 1). The term $n_k$ in (Eq. 1s') denotes the number (i.e. the cardinality of the subset $\Gamma_k$) of tasks allocated to processor $\pi_k$, and the variable i is assumed to range over the tasks of $\Gamma_k$.

$$M_k^P = \sum_{i=1}^{n_k} \max_{j=1}^{m(i)} MI_{i,j} \qquad \text{(Eq. 1s')}$$

The total memory requirements $M^P$ of the entire set $\Gamma$ of tasks is determined by adding the results found for each subset of tasks; see (Eq. $1m^{fix}$).

$$M^P = \sum_{k=1}^{p} M_k^P \qquad \text{(Eq. } 1m^{fix}\text{)}$$

When $M^P$ does not exceed the available memory $M_{sys}$, there is no need to constrain the scheduling of the tasks on any of the processors. When $M^P$ does exceed $M_{sys}$, we may constrain the scheduling of one or more tasks on one or more processors. The effect of constraining the scheduling of all tasks on a single processor can be determined using Eq. 2s′), which is a variant of (Eq. 2s).

$$M_k^D = \sum_{i=1}^{n_k} \max_{j=1}^{m(i)} MP_{i,j} + \max_{1 \le i \le n_k} (\max_{j=1}^{m(i)} MI_{i,j} - \max_{j=1}^{m(i)} MP_{i,j}) \quad \text{(Eq. 2s′)}$$

The effect of constraining the scheduling of all tasks on all processors can be determined by (Eq. $2m^{fix}$), where the total memory requirements $M^D$ is the sum of the memory requirements $M_k^D$ of each set $\Gamma_k$.

$$M^D = \sum_{k=1}^{p} M_k^D \quad \text{(Eq. } 2m^{fix}\text{)}$$

Clearly, there are many intermediate alternative embodiments, such as constraining all the tasks on just a subset of the processors, and constraining only a subset of tasks on a subset of the processors.

Alternatively, every task may be executed on every processor 460.1 - 460.3, and scheduling of the tasks is performed by the control and allocation logic module 451. The maximum memory requirements $M^P$ for the non-constrained mode remains the same, i.e. can be determined using (Eq. 1s)

$$M^P = \sum_{i=1}^{n} \max_{j=1}^{m(i)} MI_{i,j} \quad \text{(Eq. 1s)}$$

The constrained mode (i.e. all tasks are only preempted at their preemption points) requires a variant of (Eq. 2s)

$$M^D = \sum_{i=1}^{n} \max_{j=1}^{m(i)} MP_{i,j} + \max_{i=1}^{3} (\max_{j=1}^{m(i)} MI_{i,j} - \max_{j=1}^{m(i)} MP_{i,j}) \quad \text{(Eq. 2s)}$$

$$= 0.2 + 0.2 + 0.1 + \max(0.6 - 0.2, 0.8 - 0.2, 0.3 - 0.1)$$

$= 0.5 + 0.6 = 1.1$ Mbytes.

because p tasks may now run in parallel on p processors. The total memory requirements $M^D$ is now given by (Eq. $2m^{var}$).

$$M^D = \sum_{i=1}^{n} \max_{j=1}^{m(i)} MP_{i,j} + \max_{1 \le i_1 < i_2 < \ldots < i_p \le n} \sum_{k=1}^{p} (\max_{j=1}^{m(i_k)} MI_{i_k,j} - \max_{j=1}^{m(i_k)} MP_{i_k,j}) \quad \text{(Eq. } 2m^{var}\text{)}$$

Note that (Eq. $2m^{var}$) assumes $n \geq p$, and that (Eq. $2m^{var}$) is also only valid for $n \geq p$. For the special case $n < p$, the equation can be simplified to

$$M^D = \sum_{i=1}^{n} \max_{j=1}^{m(i)} MI_{i,j}$$

Clearly, there are many intermediate embodiments, such as constraining only a subset of tasks.

As an example, consider the case of two processors (i.e. $p = 2$), and three tasks (i.e. $n = 3$). It is assumed that the tasks have the same characteristics (i.e. memory requirements) as described in Tables 1 and 2 above, and their accompanying discussion. The maximum memory requirements $M^P$ for the non-constrained mode remains the same, i.e. can be determined using (Eq. 1), and hence exceeds the available memory $M_{sys}$. Using (Eq. $2m^{var}$) the maximum memory requirements $M^D$ for memory-based preemption is:

$$\begin{aligned} M^D &= \sum_{i=1}^{3} \max_{j=1}^{m(i)} MP_{i,j} + \max_{1 \le i_1 < i_2 \le 3} \sum_{k=1}^{2} (\max_{j=1}^{m(i_k)} MI_{i_k,j} - \max_{j=1}^{m(i_k)} MP_{i_k,j}) \\ &= 0.2 + 0.2 + 0.1 + \max(0.4 + 0.6, 0.4 + 0.2, 0.6 + 0.2) \\ &= 0.5 + 1.0 = 1.5 \text{ Mbytes} \end{aligned}$$

The memory requirement is lower than the available memory, meaning that provided that the tasks are preempted only at their preemption points, all three tasks can be executed concurrently.

Assume there are s pair-wise disjoint subsets $P_1, \ldots, P_s$ of processors. Let every task be allocated to a particular subset $P_l$ of processors, where $1 \leq l \leq s$. The set of tasks may therefore

be divided in s pair-wise disjoint subsets $\Gamma_1, \ldots, \Gamma_s$ of tasks. For ease of presentation, the tasks in subset $\Gamma_l$ are denoted by $\tau_i$, where $1 \leq i \leq n_l$, i.e. the tasks per subset of tasks are numbered. Similar to the variable allocation described above, the maximum memory requirements $M_l^P$ for subset $P_l$ for the non-constrained mode can be determined using (Eq. 1s″).

The maximum amount of memory required by the subset of tasks $\Gamma_l$ in the non-constrained mode executed by the subset $P_l$ of processor $\pi_k$ is given by (Eq. 1s″), which is a variant of (Eq. 1).

$$M_l^P = \sum_{i=1}^{n_l} \max_{j=1}^{m(i)} MI_{i,j} \qquad \text{(Eq. 1s″)}$$

Note that (Eq. 1s″) is similar, but not identical, to (Eq. 1s′). Whereas l ranges over subsets of processors in the former equation, k ranges over the processors in a subset in the latter equation. The total memory requirements $M^P$ for all s subsets of processors can be found by taking the sum of the requirements for the individual subsets:

$$M^P = \sum_{l=1}^{s} M_l^P \qquad \text{(Eq. 1m}^{mix}\text{)}$$

When $M^P$ exceeds $M_{sys}$, the scheduling is constrained of one or more tasks of one or more subsets of tasks executed on their associated subsets of processors. The effect of constraining the scheduling of all $n_l$ tasks of $\Gamma_l$ on a single subset $P_l$ of $p_l$ processors can be determined using (Eq. $2m^{mix}$), which is similar to (Eq. $2m^{var}$):

$$M_l^D = \sum_{i=1}^{n_l} \max_{j=1}^{m(i)} MP_{i,j} + \max_{1 \leq i_1 < i_2 < \ldots < i_p \leq n_l} \sum_{k=1}^{p_l} \left( \max_{j=1}^{m(i_k)} MI_{i_k,j} - \max_{j=1}^{m(i_k)} MP_{i_k,j} \right) \qquad \text{(Eq.2m}^{mix}\text{)}$$

Similarly to (Eq. $2m^{var}$), (Eq. $2m^{mix}$) only holds for $n_l \geq p_l$.

The effect of constraining the scheduling of all tasks of every subset of tasks on every subset of processors can be determined by (Eq. $2m^{mix'}$), where the total memory requirements $M^D$ is the sum of the memory requirements $M_l^D$ of each subset of processor $P_l$.

$$M^D = \sum_{l=1}^{s} M_l^D \qquad \text{(Eq. } 2m^{mix'}\text{)}$$

As for the fixed allocation and variable allocation cases described above, for mixed allocation there are many intermediate solutions between the non-constrained mode and the memory-based preemption mode in which all tasks are only preempted at their preemption points.

While the preferred embodiments of the present invention have been illustrated and described, it will be understood by those skilled in the art that various changes and modifications may be made, and equivalents may be substituted for elements thereof without departing from the true scope of the present invention. In addition, many modifications may be made to adapt the teaching of the present invention to a particular situation without departing from its central scope. Therefore it is intended that the present invention not be limited to the particular embodiments disclosed as the best mode contemplated for carrying out the present invention, but that the present invention include all embodiments falling within the scope of the appended claims.